Translating Lambda Calculus into C++ Templates

Vít Šefl
Charles University
Prague, Czechia

Template Metaprogramming

- **Templates facilitate parametric polymorphism**
  - Template declaration abstracts over one or more types
  - Concrete types are substituted when the template is instantiated
- **More expressive than intended**
  - Can be used to perform arbitrary computations
  - Templates are only present during compilation
  - Enables compile-time computations

Example Metaprogram

```
template <bool, typename, typename>
struct if_;

template <typename T, typename F>
struct if_<true, T, F>
{ using type = T; };

template <typename T, typename F>
struct if_<false, T, F>
{ using type = F; };


if_<1 != 2, int, char>::type five = 5;
```

Practical Uses

- Compile-time optimizations
  - Moving computations from run-time to compile-time
  - Less relevant thanks to `constexpr`
- Generic programming
  - Type manipulation, type traits
  - `std::enable_if`, `std::conditional`, `std::is_same`, etc
  - Typically in library code

Problems: Boilerplate Code

```cpp
template <template <typename> class, typename>
struct filter;

template <template <typename> class F>
struct filter<F, list<>>
{
  using type = list<>;
};

template <template <typename> class F, typename A, typename... R>
struct filter<F, list<A, R...>>
{
  using type = typename std::conditional<
    F<A>::value,
    typename cons<A, typename filter<F, list<R...>>::type>::type,
    typename filter<F, list<R...>>::type
  >::type;
};
```

Problems: Boilerplate Code

```
template <template <typename> class, typename>
struct filter;

template <template <typename> class F>
struct filter<F, list<>>
{
  using type = list<>;
};

template <template <typename> class F, typename A, typename... R>
struct filter<F, list<A, R...>>
{
  using type = typename std::conditional<
    F<A>::value,
    typename cons<A, typename filter<F, list<R...>>::type>::type,
    typename filter<F, list<R...>>::type
  >::type;
};
```

Problems: Error Messages

```
error.cpp:31:3: error: type/value mismatch at argument 2 in template parameter list for
'template<bool <anonymous>, class, class> struct std::conditional'
   31 |     >::type;
      |     ^
error.cpp:31:3: note:   expected a type, got 'cons<A, typename filter<F, list<R ...>
>::type>::type'
error.cpp: In function 'int main()':
error.cpp:48:52: error: 'type' in 'struct filter<is_good, list<char, int> >' does not
name a type
   48 |     using result = filter<is_good, list<char, int>>::type;
      |                                                      ^~~~
```

- Actual error?
  - Missing typename

Solutions

- **Metaprogramming frameworks**
  - e.g. Boost.Hana
  - Reduce the amount of boilerplate
  - Clearer error messages with `static_assert`
- **Translation**
  - e.g. EClean
  - Write metaprograms in a different language
  - Avoid boilerplate and error messages

Translation

- **Template metaprogramming is a form of functional programming**
  - Template can be seen as a type-level function
  - Data is immutable
  - Computation relies on recursion
- **Use a functional language to describe metaprograms**
  - Translation into C++ templates
  - Type system can turn template compilation errors into type errors

Goals

- **Use lambda calculus as the source language**
- **Add common functional features**
  - Algebraic data types, recursion, etc
- **The translation should be:**
  - *Simple* – easy to understand and maintain
  - *Extensible* – able to incorporate existing metaprograms
  - *Lazy* – avoid unnecessary template instantiation

Lambda Calculus Translation

```
translate(x) :=
  using type = typename x::type;

translate(\x → M) :=
  struct type {
    template <typename x>
    struct apply {
      translate(M)
    };
  };

translate(M N) :=
  struct S1 { translate(M) };
  struct S2 { translate(N) };
  using type = typename S1::type::template apply<S2>::type;
```

Translation Example

- Input

```
succ  = \x -> x + 1
twice = \f x -> f (f x)
five  = twice succ 3
```

- Output

```
struct succ  { ... };
struct twice { ... };
struct five  { ... };

five::type        == Int<5>
five::type::value ==     5
```

Lazy Evaluation

- Input

```
k1 = \x y → x
k2 = \x y → y
ap = \f → f 1 (let x = x in x)
```

- Output

```
struct k1 { ... };
struct k2 { ... };
struct ap { ... };

ap::type::apply<k1>::type == Int<1>
ap::type::apply<k2>::type == ⊥
```

Lazy Evaluation

```
...
loop.cpp:119:7:   [ skipping 889 instantiation contexts, use -ftemplate-backtrace-limit=0 to disable ]
...
loop.cpp:119:7: fatal error: template instantiation depth exceeds maximum of 900 (use '-ftemplate-depth=' to increase the maximum)
  119 | using type = typename _T2::type;
      |       ^~~~
compilation terminated.
```

- Practical limit
  - Compilation should always terminate
  - Can be increased in most compilers

Algebraic Data Types

- Input

```
data List a = Nil | Cons a (List a)

countdown = \n → if n < 0 then Nil else Cons n (countdown (n - 1))
numbers   = countdown 2
```

- Output

```
template <typename, typename>
struct cons { };
struct nil { };

numbers::type == cons<Int<2>, cons<Int<1>, cons<Int<0>, nil>>>
```

Integration with Existing Metaprograms

- Simple adapters for library metaprograms

```
template <template <typename> class F>
struct predicate {
  template <typename T>
  struct apply {
    using type = Bool<F<typename T::type>::value>;
  };
};
```

- Example use

```
using is_pointer  = predicate<std::is_pointer>;
using filter_ptrs = filter::type::apply<is_pointer>;
```

Type Systems

- Source language can optionally have a type system
- Compare

```
bad = 5 5
```

  - Possible type error

```
src:1:7: 5 :: Int is not a function
```

  - Possible compile error

```
src: In instantiation of 'struct bad':
src:24:33:   required from here
src:17:9: error: no class template named 'apply' in 'using type = struct
Int<5>' {aka 'struct Int<5>'}
   17 |    using type = typename s1::type::template apply<s2>::type;
      |          ^~~~
```

Summary

- Increasing support for compile-time computations in recent C++ standards
  - `constexpr`
- Support for type-level programming still lacking
  - Verbose and error-prone
- New method of metaprogram translation
  - Focus on simplicity and well-defined, non-strict semantics
  - Can be used with macros, higher-order metaprograms, and third-party tools
  - Type agnostic

Thank you for your attention